Cloudproxy Nuts and Bolts

manferdelli@, tmroeder@

Overview

Cloudproxy is a software system that provides *remotely authenticated* isolation, confidentiality and integrity of code and data for Hosted Systems preventing attacks from co-tenants and (under modest assumptions) insiders in a remote data center on supporting hardware. To achieve this, Cloudproxy uses on two components: a “Host System” (raw hardware, Virtual Machine Manager, Operating System) which provides capabilities described below to a “Hosted System” (VM, Application, Container).

Cloudproxy provides a mechanism, at each level of the software stack, to isolate Hosted Systems, measure and remotely verify the exact software and configuration information constituting the Hosted System and provide security services like sealing that ensures that information (like keys) can be securely provisioned and retrieved only by the correct Hosted System, while isolated, on a supported platform.

A key concept for Cloudproxy is Code Identity and Measurement that is coupled with isolation and secret provisioning. A Host System measures a Hosted System incorporating the actual binary code and configuration information affecting execution resulting in an unforgeable, compact global identity for that code and execution context. Since the Hosted System knows the “identity” of each Hosted System (i.e.- the unforgeable global identity), it can store secrets that only the Hosted System will receive[1]. The Host Systems can also “attest” to statements made by Hosted Systems by incorporating the unforgeable global identity in statements it signs (again with keys only an isolated Host System has access to). The upshot of this is that a Cloudproxy Hosted System can be isolated, maintain secrets only it knows to encrypt and integrity protect all data it receives or sends, and it can securely authenticate itself over an otherwise unprotected network connection and thus employ authenticated public keys tied to its identity that can be relied upon by communicating parties.

Readers can consult [1] for a fuller description.

The Tao

A Hosted System uses the Cloudproxy API, called the Tao, to achieve the security promises (program isolation, and confidentiality and integrity for programs and data) provided by Cloudproxy. The programming model is simple and require only a few API calls. The Tao Library is linked into an executable to provide the programming interface in Go or C++.

The basic calls are:

[1] To do this, the Host System must be isolated and have access to secrets only it knows. The foundation for this consists of primitives hardware provides to the “base” Cloudproxy systems it boots.

AddHostedProgram instructs the Host System to measure and start a new, isolated Hosted System. It names the binary image and other context data to start the program. The Hosted System could be, for example, a VM if the Host System is a VMM or an isolated Linux process if the Host System is Linux.

Seal takes an opaque data blob and appends the measurement of the Hosted System. It encrypts and integrity protects the resulting object (using keys only the Host System knows) and returns the resulting opaque object to the Hosted System. Hosted Systems typically "seal" private signing and encryption keys so they can be later recovered when the Hosted System is restarted using "Unseal" below.

Unseal takes an opaque blob (produced by a prior "Seal") from a Hosted System. It decrypts (and checks the integrity of) the blob and compares the measurement of the Hosted System requesting the unseal with the measurement of the Hosted System named in the blob. If the measurements match, it returns the protected data.

Attest takes a blob from a Hosted System and signs a statement naming the blob and the measurement of the Hosted System requesting the Attest. It returns the signed statement (the "Attestation") along with a certificate (the "Host Certificate") from an authority certifying that the public key it used to sign the statement belongs to a verified Host System with enumerated security characteristics. See [1] for details for the "Trust Model" enabling a recipient of such a certificate to rely on the association between the public key named in the Host Certificate and a trustworthy Cloudproxy Hosted System. The meaning of the signed blob is, informally, "Statement X came from the program with Measurement M while it was isolated." Hosted Systems mainly use attest as follows: The Hosted System generates a public-private key pair and seal the private key. Then they request an Attestation of the corresponding public key. A party receiving the Attestation and Host Certificate can cryptographically verify the public key came from the named program while isolated and thus subsequent proof of possession of the private key can be used to authenticate statements from the Hosted System.

GetRandom provides cryptographically random bits, typically for key generation.

Principal Names

Principal names in Cloudproxy are hierarchical and securely name the principal. For example, a principal rooted in a public key will have the public key in its name and a program principal (a measured Cloudproxy Hosted System) will have the measurement in the principal name (i.e.-a cryptographic hash).

The root name for a hosted program, in the development case, might look something like

```
key([08011001180122450801124104 6cdc82f70552eb...]).Program([25fac93bd4cc868352c78f4d34df6
d2747a17f85...])
```

Here, `key([080110011801224508011…])` represents the signing key of the host and `Program([25fac93bd4cc868352c78f4d34df6d2747a17f85...])` extends the host name with the hash of the Hosted System. If the host were a Linux host rooted in a TPM boot, its name would name the AIK and the PCRs of the booted Linux systems, the hash of the Authenticated Code Module ("ACM") that initiated the authenticated boot and the hash of the Linux image and it's initramfs[2].

The Tao Paradigm

The Tao is almost always used in a stereotypical way which we refer to as the Tao Paradigm. Programs always have policy public keys embedded (PK_{policy}) in their image either explicitly or implicitly. Statements signed by the corresponding private key (pK_{policy}), and only those statements, are accepted as authoritative and acted on by these programs. The policy key(s) plus the Hosted System code and configuration, reflected in its measurement, fully describe how the Hosted System should behave and, hence, an authenticated measurement is a reliable description of expected behavior.

In the Tao Paradigm, when a program first starts on a Hosted System, it makes up a public/private key-pair ($PK_{program}$/ $pK_{program}$) and several symmetric keys that it uses to "seal" information for itself. A Hosted System then "seals," using the Host System interface, all this private (key) information[3]. Next it requests an Attestation from its Host System, naming the newly generated $PK_{program}$ and sends the Attestation to a service for the security domain which confirms the security properties in the Attestation and Host Certificate[4]. If the Attestation and Host Certificate meet security domain requirements, the security domain service signs (with pK_{policy}) an x509 certificate specifying $PK_{program}$ and the Tao Principal Name of the Hosted System (as mentioned above, this name, specifies, among other things, the Hosted System measurement). The resulting certificate, called the Program Certificate, can be used by any Hosted System to prove its identity to another Hosted System in the same security domain. Program Certificates are used to negotiate encrypted, integrity protected SSL-like channels between Hosted Systems (the "Tao Channel") and a Hosted System can share information over these channels with full assurance of the code identity and security properties of its channel peer. Once established, each endpoint of the Tao Channel "speaks for" its respective Hosted System.

Hosted Systems in the same security domain can fully trust other Hosted Systems in the same security domain with data or processing. Typically, a Hosted System uses the symmetric keys it generates and seals at initialization to encrypt and integrity protect information it stores on disks or remotely.

[2] Initramfs will have security critical code like the service that implements the Tao so it must be measured along with the kernel image to provide an accurate identity for the "running Linux OS."
[3] The Tao also provides rollback protection for this sealed data.
[4] The actual attestation being signed by the Host System expressed in a formalized language is $PK_{program}$ speaksfor the Hosted System Principal name.

Employing a centralized security domain service eliminates the need for each and every Cloudproxy Hosted System in a security domain to maintain lists of trusted hardware or trusted programs and simplifies distribution, maintenance and upgrade.

Often, Hosted Systems in the same security domain will share intermediate keys used to protect data that may be used on many Host System environments. As discussed below, when software is upgraded or a new Hosted System in a security domain is added, these keys can be shared based on policy-key signed directives as Host or Hosted Systems are upgraded or new systems are introduced in a controlled but flexible way eliminating the danger that data might become inaccessible if a particular Cloudproxy system is replaced or becomes damaged or unavailable.

The Extended Tao

Given the Tao Paradigm that is almost universally employed, the Tao contains some additional support functions. All these functions are in the tao package (along with the basic tao functions above and can be called, for example as *tao.StartHostedProgram*, etc.

DomainLoad is used to store and retrieve Program Certificates and sealed data.

GetTaoName

GetSharedSecret

Parent

ExtendTaoName allows a Hosted System to extend its Principal Name with arbitrary data. For example, rather than having a policy embedded in a program image, a Hosted System can extend its name with a policy key it reads and the new Principal Name will reflect this value.

There are helper functions to build and verify Program Certificates, perform common crypto tasks like key generation and establish the Tao Channel.

There is also support for authorization in the Tao. Guards make authorization decisions. Current guards include:

- the liberal guard: this guard returns true for every authorization query
- the conservative guard: this guard returns false for every authorization query
- the ACL guard: this guard provides a list of statements that must return true when the guard is queried for these statements.
- the datalog guard: this guard translates statements in the CloudProxy auth language (see tao/auth/doc.go for details) to datalog statements and uses the Go datalog engine from github.com/kevinawalsh/datalog to answer authorization queries. See install.sh for an example policy.

Hardware roots of Trust

Cloudproxy requires that the lowest level system software be measured by a hardware component which must also be able to provide attest services and seal/unseal services (and optionally some hardware assist to isolate of Hosted Systems). Absent hardware protection, remote users have no principled way to trust the security promises (isolation, confidentiality, integrity, verified code identity) since “insiders” might silently change security critical software or steal keys.

Cloudproxy supports TPM 1.2 and TPM 2.0 as hardware roots of Trust for Host Systems booted on raw hardware. We have implemented support for other mechanisms. Adding a new hardware mechanism is relatively straightforward provided the new hardware supports accurate measurement of booted images, isolation for Hosted Systems and attestation which is used to initialize the key hierarchy.

Once the base Host System is safely measured and booted on a supported hardware, Cloudproxy implements support for recursive Host Systems at almost every layer of software including:

1. A Host System consisting of hardware (e.g. - TPM, SMX) that hosts a VMM which isolates Hosted Systems consisting of Virtual Machines.
2. A Host System running in an operating system which isolates Hosted Systems consisting of processes (or applications).
3. A Host System running in an operating system which isolates Hosted systems hosted consisting of subordinate Operating Systems or Containers.
4. A Host System running in an application (like a browser) which isolates Hosted Systems consisting of sub-applications, like plug-ins.

In all cases, Hosted Systems have the same Tao interface to the Host System and can use any non-Cloudproxy host service (for example, any system call on Linux) so the programming model at each Hosted System layer is essentially unchanged from the non-Cloudproxy case.

Sample Applications

This paper is intended to allow you to use Cloudproxy immediately on a Linux based Cloudproxy Host System. To this end we include installation instructions for TPM 2.0 protected hardware with SMX extensions and a complete annotated simple application called, cleverly, SimpleExample.

There are more complex examples in go/apps.

Installing Cloudproxy

Complete instructions for Linux installation which allows you to run Simple application are here.

When your ready for installation instructions for a VMM, look at the installations instructions for KVM here

Simple Example

We describe annotated sample code for a simple example in Go and C++ containing all the critical Cloudproxy elements. A full working version of the example is in cloudproxy/go/apps/simpleexample for the Go version and cloudproxy/simpleexample for the C++ version. Since the domain service does not use Tao primitives directly, we don't annotate that code here although go/apps/simpleexample contains a full working version.

You have a correctly installed Go development tools or C++ development tools as well as protobuf, gtest and gflags and others to compile and run SimpleExample. You can do this as follows:

```
go get -u github.com/golang/protobuf/{proto,protoc-gen-go}
go get github.com/golang/crypto/...
go get github.com/golang/glog
go get github.com/golang/lint/golint
go get code.google.com/p/gcfg
go get github.com/google/go-tpm/tpm
go get github.com/jlmucb/cloudproxy/tao/auth
go get github.com/jlmucb/cloudproxy/util
go get github.com/jlmucb/cloudproxy/util/protorpc
go get github.com/kevinawalsh/datalog/dlengine
go get code.google.com/p/go.text/encoding
go get code.google.com/p/google-api-go-client/compute/v1
```

There are three application components in SimpleExample, each producing a separate executable:

1. Simple Client (in simpleclient.go)
2. Simple Server (in simpleserver.go)
3. Simple Security Domain Signing Service (in simpledomainservice.go)

Common code used by the client and server is in simplecommon.go.

The Simple Server makes up a secret and stores it. The Simple Client uses a Tao Channel to contact the Simple Server to learn the secret. We don't implement rollback protection or distributed key management for intermediate secrets in SimpleExample just to keep the example as simple as possible. The sample application also includes a Simple Security Domain Signing Service which checks the measurements in the Attestations for the Simple Client and Simple Server and, if the measurements are correct, signs the Program Certificate.

In the annotated code in this document, error codes and some helper functions omitted.

The API – Go

Domains represent security contexts. Security contexts encapsulate configuration information like names, path to key blobs, path to policy key, and the guard employed for authorization decisions.

CreateDomain initializes a new Domain, writing its configuration files to a directory. This creates the directory and, if needed, a policy key pair encrypted with the given password when stored on disk; it also initializes a default guard. The call is:

```
        func CreateDomain(cfg DomainConfig, configPath string, password []byte) (*Domain, error)
```

Any parameters left empty in cfg will be set to reasonable default values.

Domain information is loaded from a text file, typically called tao.config via the call:

```
        LoadDomain(configPath string, password []byte)(*Domain, error)
```

which returns a domain object, if successful. The password is used to load a key set from disk. If no password is provided, then LoadDomain will attempt to load verification keys only. For example, LoadDomain is called with a configPath and an nil password to load the policy verification key.

A configuration object, type DomainConfig, holds configuration information for the domain between Tao activations.
The network interface for the Tao channel consists of:

- `func DialWithKeys(network, addr string, guard tao.Guard, v *tao.Verifier, keys *tao.Keys) (net.Conn, error)`
- `func Listen(network, laddr string, config *tls.Config, g tao.Guard, v *tao.Verifier, del *tao.Attestation) (net.Listener, error)`

The API -- C++

Simple Client in Go (annotated)

Simple Server in Go

Some Common code in Go

Simple Client in C++

Simple Server in C++

Running SimpleExample

When the tao starts on a Host System, it requires three kinds of information:

- A public key that roots the *Tao* on the hardware,
- Host data,

- Domain data (in our case for the simpleexample domain) including the policy key and corresponding private key, for example the policy key and certificate.

In addition, we need an implementation for the "Host System." In our case, the Host System is Linux and the implementation (whether using a soft tao or a TPM) is linux_host.

The public key rooting the hardware tao is usually produced by a tpm utility; in the TPM 1.2 nomenclature, this is called the AIK. The public key rooting the TPM 2.0 is the endorsement key. In our demo, we use a "soft tao" which is rooted in a key.

The Host Data consists of…

The Domain data (in our case for the simpleexample domain) including the policy key and corresponding private key, hostname, and information related to the guards used[5] as well as signatures over the binaries that are part of the domain. In our case, these are the simpleclient and simpleserver binaries.

In simpleexample, all this information is in SimpleDomain/domain.simpleexample. Other sub-directories of SimpleDomain/domain.simpleexample, namely, SimpleClient, SimpleServer and SimpleDomainService store files containing data (like sealed keys and Program Certificates), for these programs.

There is a single utility, called *tao* which initializes this domain data, activates the tao host and runs the applications. We provide shell scripts to call *tao* with the right arguments, these scripts are in SimpleDomain.

The scripts use several path variables, namely:

```
TAO_HOST_DIR=~/src/github.com/jlmucb/cloudproxy/go/apps/simpleexample/SimpleDomain
TEMPLATE=$TAO_HOST_DIR/domain_template.simpleexample
DOMAIN=$TAO_HOST_DIR/domain.simpleexample
BINPATH=~/bin
```

In addition, we need a generic domain template. We have provided a sample template in SimpleDomain/domain_template.simpleexample. However, you can generate such a template by running gentemplate, which consists of:

```
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -pass "xxx"
/home/jlm/src/github.com/jlmucb/cloudproxy/go/run/scripts/domain_template.pb > $TEMPLATE
sed "s/REPLACE_WITH_DOMAIN_GUARD_TYPE/Datalog/g"
```

This template contains information included in the policy cert, the basic datalog rules used by the domain when authenticating images and the location of the images which must me measured and included in the policy database in SimpleDomainService.

To initialize the (soft) key, call initkey which does the following:

```
$BINPATH/ tao domain newsoft -soft_pass xxx -config_template $TEMPLATE
     $DOMAIN/linux_tao_host" >> $TEMPLATE
```

"`newsoft`" means generate a new soft key. The arguments following the flags "`-config_template -tao_ -pass`" specify respectively the location of the template, the location where the domain information is stored and the password protecting the private policy key. This produces the xxx file containing root Tao key.

[5] Look at domain.go for further details.

To initialize the domain, call initidomain which does the following:

```
$BINPATH/tao domain init -tao_domain $DOMAIN -config_template $TEMPLATE -pub_domain_address
"1.2.3.4" -pass xxx
$BINPATH/tao domain policy -add_host -add_programs -add_linux_host -add_guard -tao_domain \
      $DOMAIN -pass xxx -config_template $TEMPLATE
```

The first call produces the files in $DOMAIN/linux_tao_host/{cert,keys,host.config}.

To initialize the (Linux) host, call inithost which does the following:

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -root -pass xxx
```

This generates linux host configuration information which is in SimpleDomain/domain.simpleexample/linux_tao_host. The argument to the "`-hosting`" flag is the kind of child hosts, namely, Linux processes. The "`-root`" flag means this is a "root" host (i.e. – the lowest level tao). For hosts stacked on other hosts, we would use the "`-stacked`" flag. For example,

```
$BINPATH/tao host init -tao_domain $DOMAIN -hosting process -stacked -parent_type tpm
```

To run the host, call runhost, which consists of:

```
$BINPATH/tao host start -tao_domain $DOMAIN -host linux_tao_host/ -pass xxx &
```

The argument to the "`-host`" flag is the subdirectory of SimpleDomain/domain.simpleexample that contains the host information.

Finally, to run a Hosted System, like simlpeclient, we would say:

```
$BINPATH/tao run $BINPATH/simpleclient -tao_domain $DOMAIN –pass xxx &
$BINPATH/tao host stop -tao_domain $DOMAIN
```

We have provided an additional script, "runall" which starts all the Hosted Systems and SimpleDomainService.

So to run simpleexample the very first time, call initkey, initdomain and inithost. If no host is running, call runhost. Each time you run tests call runall but remember to kill these services afterwards. To list Hosted Programs currently running call:

```
tao list …
```

To kill hosted programs call

```
tao kill …
```

Upgrade and key management scenarios

Since sealed material is only provided to a Hosted System with exactly the same code identity that sealed the material running on the exact same Host System, while isolated by that Host System, you may be worried about lost data when a Hosted System breaks or becomes unavailable or limitations that affect key management, software upgrade or distribution when the Hosted System runs on other Host Systems. In fact, it is rather easy to accommodate all these circumstances, and many others, efficiently, securely and in most cases automatically using Cloudproxy, although the Cloudproxy applications must make provisions for this during development.

Below are a few sever example key management techniques that can be used when a Cloudproxy application is upgraded, a new Cloudproxy application (in the same security domain) is launched, or as applications migrate to other Host Systems. All these mechanisms preserve the confidentiality and integrity of all Cloudproxy applications and their data.

There is a discussion of many of the mechanisms, as they might affect client software used across different security domains, by users with no control over the application code while supporting consumer transparency (the most challenging case) in [4]. Here we restrict ourselves to cooperating server applications for simplicity.

To ease description, imagine all application data is stored locally or remotely and probably redundantly in encrypted, integrity protected files. Each file is encrypted and integrity protected with individual file keys and each file key is itself encrypted and integrity protected with a partitioned Sealing keys. Different partitions are protected by different keys to reduce the risk of universal compromise. Every key has exposed meta data consisting of a globally unique name for the entity it protects, the key type and an "epoch." Epochs increases monotonically as the keys are rotated[6]. As keys for a new epoch become available, the objects they protect are re-encrypted, over a reasonable period of time (the Rotation Period). During this time, keys for the prior epoch are available and can be used to decrypt objects; however, as soon as new epoch keys are available, all new data is encrypted with the new epoch keys. At the end of the Rotation Period, once applications have confirmed that all data is protected with the keys from the most recent epoch, old epoch keys are deprecated. As a reminder: there are other possible mechanisms to do key management.

1. The first option to deal with "brittle keys" protecting application data is standard: use a distributed key server like Keyczar (or many others). In this case, Cloudlproxy applications do not locally store data protection keys but contact a key server (over a Tao Channel). The key server (which does key rotation, etc., as many do) authenticates the Hosted System that needs keys and verifies that it is authorized to receive those keys; if so they are transmitted over the Tao Channel. Hosted Systems can be upgraded and all authorization policy can be maintained by the key service. Hosted Systems will need to respond to "reinitialize" requests periodically as keys rotate.

2. An alternative, less centralized, key rotation mechanism allows individual Hosted Systems maintaining their own keys to protect files as well as perform key rotation themselves. When software is upgraded or new programs are introduced, the new programs or upgraded programs come with a certificate signed by the policy key that instruct one Hosted System to disclose these keys to the new version (or new) Hosted System. Since this can result in lost data if a Host System becomes unavailable, Hosted Systems would likely distribute these keys to different instances on different machines to ensure continuity.

3. Finally, when new data protection keys are established for an application task, Hosted Systems can contact a domain service to receive intermediate keys for registered files or file classes. These keys can be sealed using the Host System provided Seal and used without contacting the service each time the Hosted System starts. This mechanism places additional administrative burden on each Hosted System to contact the "key sharing service" as intermediate keys rotate but this is not uncommon.

[6] And you certainly should rotate keys as part of effective cryptographic hygiene!

It is important to note that while the foregoing descriptions treat keys as “all or nothing” entities, all these scheme have corresponding “split key” implementations to achieve higher security. In addition to pure key management, any security domain may elect to have an authorized Cloudproxy Hosted System archive data. Such an archive application, upon which security domain policy confers access to data, can, in the background, archive data to (centralized or distributed) repositories. Finally, note that application upgrade (given a data key management solution) is automatic even when the policy keys change: New versions of Hosted Systems simply re-initialize (get new program keys and certificates) using the (centralized or distributed) security domain service and no special provision, aside from current policy at the security domain service, need be provided[7].

Suggested Exercises

That’s all there is to using Cloudproxy. Here are some suggested exercises to complete the training:

1. Write a more complicated set of domain applications; for example, see “go/apps/fileproxy.”
2. Boot a Linux Host System on tpm supported hardware using the TPM to root the Linux Tao (see … for instructions).
3. Boot a KVM Host System on tpm supported hardware and then run a stacked VM host in a Linux partition (see … for instructions). Simpleexample should run fine in the VM(s) with slight changes to the initialization scripts.
4. Explore the Data log engine (examples?)

References

[1] Manferdelli, Roeder, Schneider, The CloudProxy Tao for Trusted Computing, http://www.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-135.pdf.
[2] CloudProxy Source code, http://github.com/jlmucb/cloudproxy.
[3] TCG, TPM specs,
http://www.trustedcomputinggroup.org/resources/tpm_library_specification
[4] **Beekman, Manferdelli, Wagner,** AsiaCCS, 2016.

[7] Many events may cause such a policy change including a determination that previously trusted hardware elements have been compromised.

The Guards (remove this)

The Guard interface:

- `Subprincipal() auth.SubPrin`: returns a unique subprincipal for this policy.
- `Save(key *Signer) error`: writes all persistent policy data to disk, signed by key
- `Authorize(name auth.Prin, op string, args []string) error`
- `Retract(name auth.Prin, op string, args []string) error`
- `IsAuthorized(name auth.Prin, op string, args []string) bool`
- `AddRule(rule string) error`
- `RetractRule(rule string) error`
- `Clear() error`: removes all rules.
- `Query(query string) (bool, error)`
- `RuleCount() int`
- `GetRule(i int) string`.
- `String() string`: returns a string suitable for showing auth info.